Rust for C++ developers

What this talk will NOT be about

- Rust syntax
- Enums
- Error handling
- Generics
- Macros
- Cargo
- Interoperability with C++

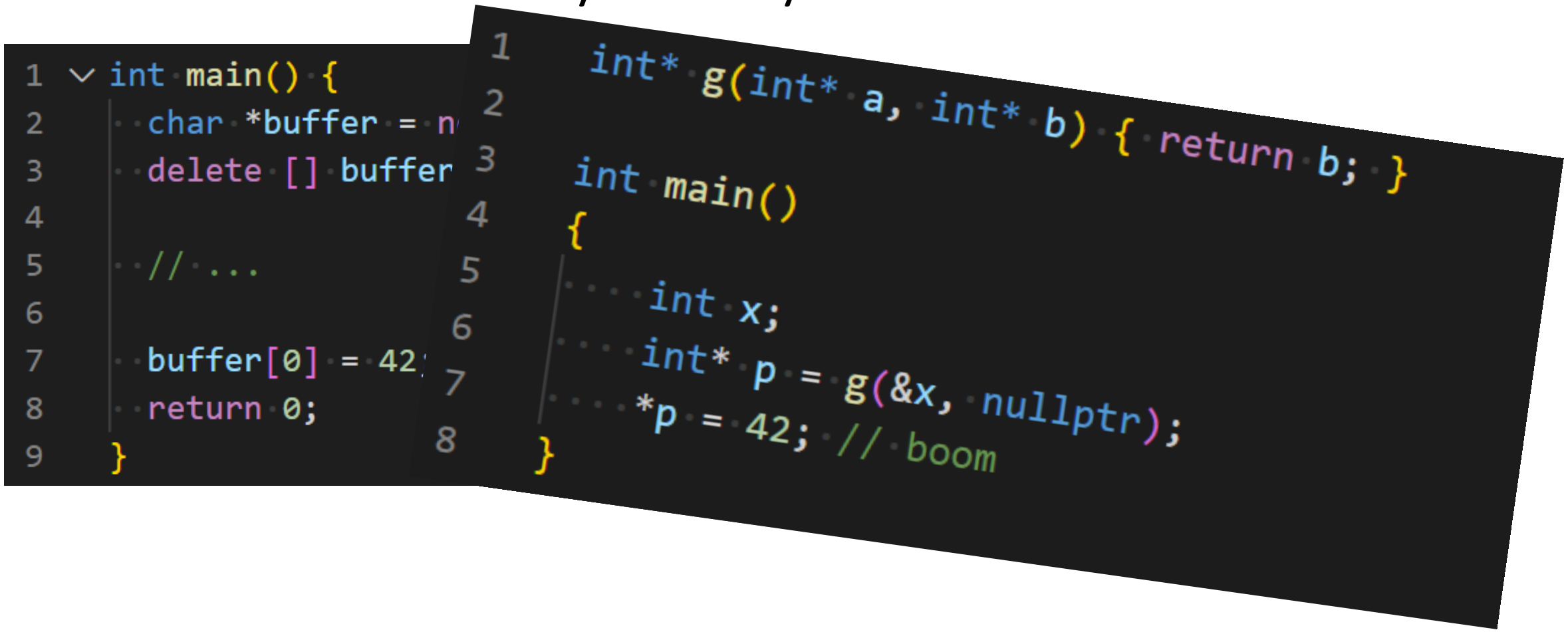
What this talk will be about

- What is memory safety
- How Rust ensures memory safety at compile time
- How to argue with the borrow checker

What is memory safety?

```
1  int main() {
2      char *buffer = new char[42];
3      delete [] buffer;
4
5      // ...
6
7      buffer[0] = 42;  // Boom!
8      return 0;
9  }
```

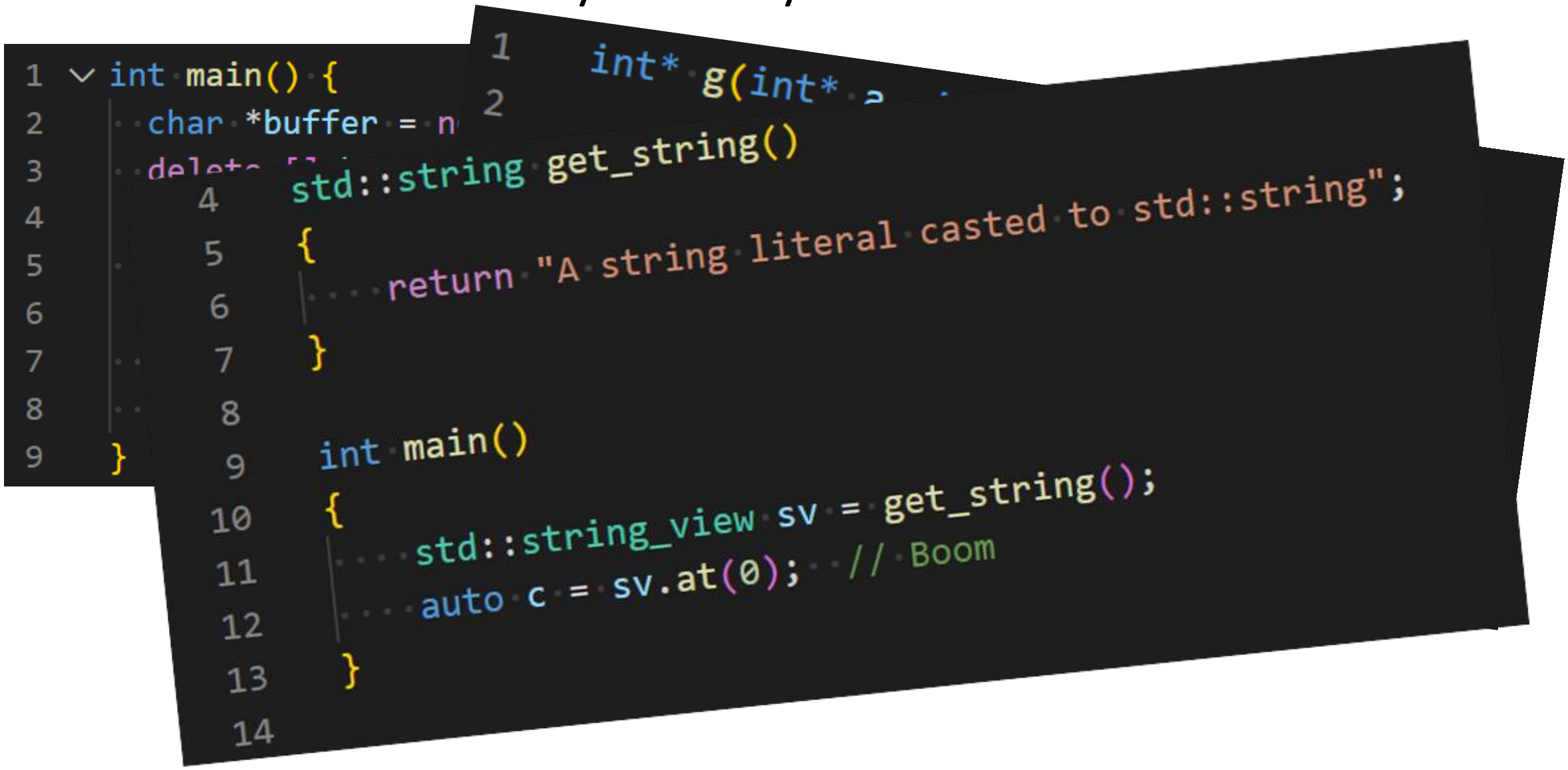
What is memory safety?



```
int main() {
    char *buffer = n
    delete [] buffer

    // ...

    buffer[0] = 42;
    return 0;
}
```

```
int* g(int* a, int* b) { return b; }

int main()
{
    int x;
    int* p = g(&x, nullptr);
    *p = 42; // boom
}
```

What is memory safety?



What is memory safety?

```cpp
int main() {
    auto v = std::vector<int>();
    v.push_back(0);
    auto first_elem = &v[0];
    v.push_back(1);
    std::cout << "First elem is: " << *first_elem << std::endl;
    return 0;
}
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
> g++ .\push_back.cpp -o push_back.exe
> .\push_back.exe
First elem is: -2011292912
>
```

RAII

- Bind the life cycle of a resource to the lifetime of an object

```
std::mutex m;

void bad()
{
    m.lock();                     // acquire the mutex
    f();                          // if f() throws an exception, the mutex is never released
    if(!everything_ok()) return; // early return, the mutex is never released
    m.unlock();                   // if bad() reaches this statement, the mutex is released
}

void good()
{
    std::lock_guard<std::mutex> lk(m); // RAII class: mutex acquisition is initialization
    f();                               // if f() throws an exception, the mutex is released
    if(!everything_ok()) return;       // early return, the mutex is released
}                                      // if good() returns normally, the mutex is released
```

RAII

- Example with memory

```
1
2
3   int main() {
4       auto vect = new int[30];
5
6       //....
7
8       delete vect;
9   }
10
```

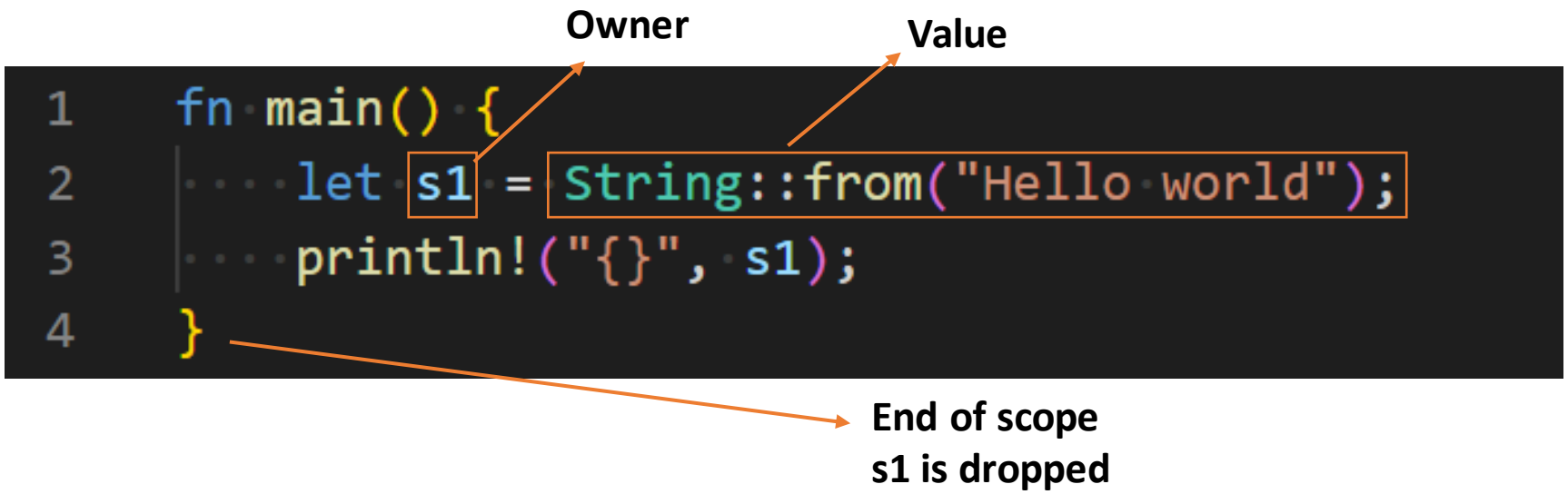
```
1   #include <vector>
2
3   int main() {
4       auto vect = std::vector<int>(30);
5
6       //....
7
8
9   }
10
```

How Rust ensures safety

- Ownership
- Borrowing
- Lifetimes

Ownership

- Each value has an owner
- There can be **only one owner** at a time
- When the owner goes out of scope, the value will be dropped (released from memory)



Ownership

	C++	Rust
a = b	- Depends on copy/move constructors - Mostly copies	- If type is Copy, copy - Otherwise, move

- Assignment
  - Primitive types or classes that implements Copy -> copy
  - Otherwise -> trasfers ownership (similar to std::move)

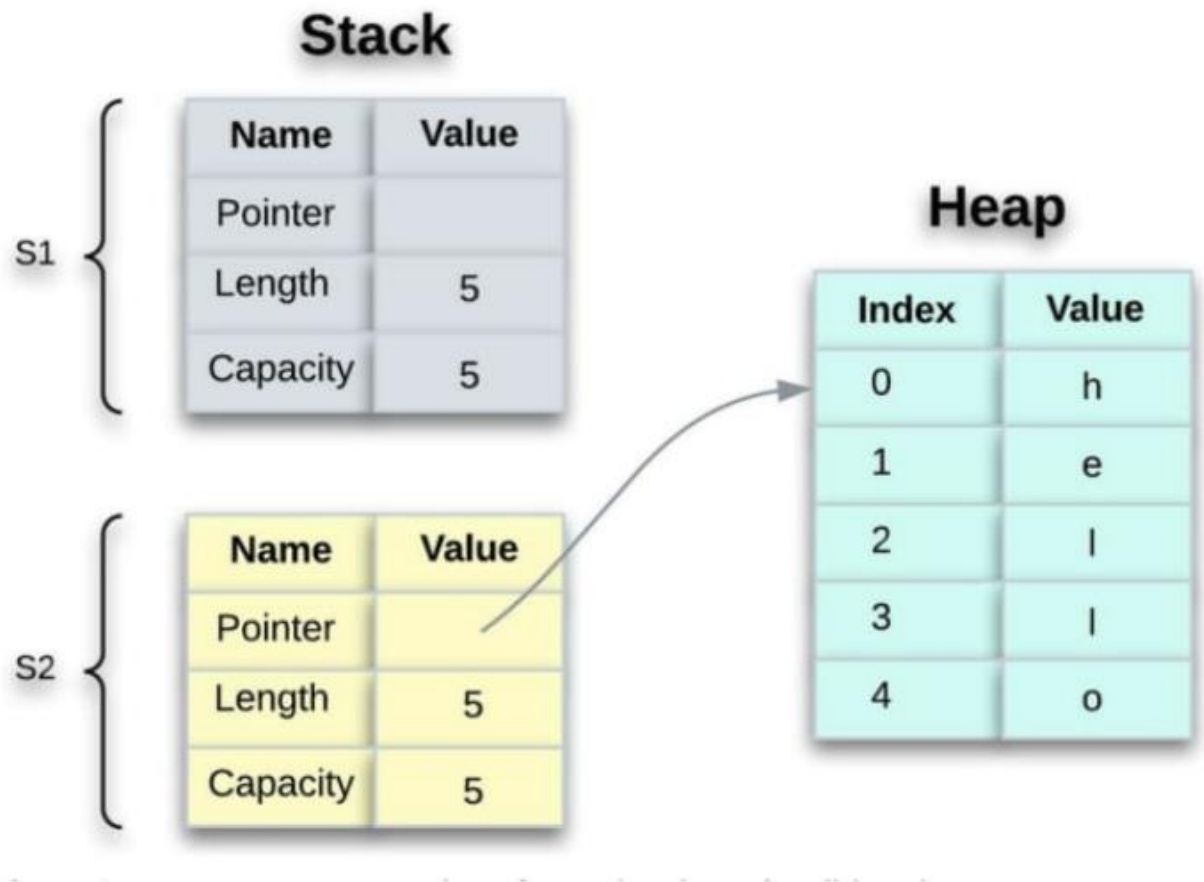
Example 1

```
let x = 5;
let y = x;
println!("{} {}", x, y);  // OK -> primitive values are copied
```

Example 2

```
let s1: String = String::from("Hello world");  // 's1' is the owner of this string
let s2: String = s1;  // Ownership is trasfered --> s2 is the new owner and s1 is dropped
println!("{} {}", s1, s2);  // Compiler error -> s1 is not valid anymore
```

Ownership


Stack
S1
Name
Value
Pointer
Length
5
Capacity
5
S2
Name
Value
Pointer
Length
5
Capacity
5
Heap
Index
Value
0
h
1
e
2
l
3
l
4
o

Ownership

Example 3

```
fn print_string(s: String) {
    println!("The string is: {}", s);
}

▶ Run | Debug
fn main() {
    let s: String = String::from("Hello world!");
    print_string(s);

    println!("{}", s);
}
```

Ownership

```
> cargo build
   Compiling ownership v0.1.0 (C:\Users\andre\Documents\Workspace\RustExperiments\ownership)
error[E0382]: borrow of moved value: `s`
 --> src\main.rs:9:20
  |
6 |     let s = String::from("Hello world!");
  |         - move occurs because `s` has type `String`, which does not implement the `Copy` trait
7 |     print_string(s);
  |                  - value moved here
8 |
9 |     println!("{}", s);
  |                    ^ value borrowed here after move
  |
  = note: this error originates in the macro `$crate::format_args_nl` which comes from the expansion of the macro
 `println` (in Nightly builds, run with -Z macro-backtrace for more info)

For more information about this error, try `rustc --explain E0382`.
error: could not compile `ownership` due to previous error
>
```

Borrowing

	C++	Rust
Mutable reference	auto a = &b	let a = &**mut** b
Immutable reference	**const** auto a = &b	let a = &b

- You can have immutable references

```
let s1: &String = &s;
```

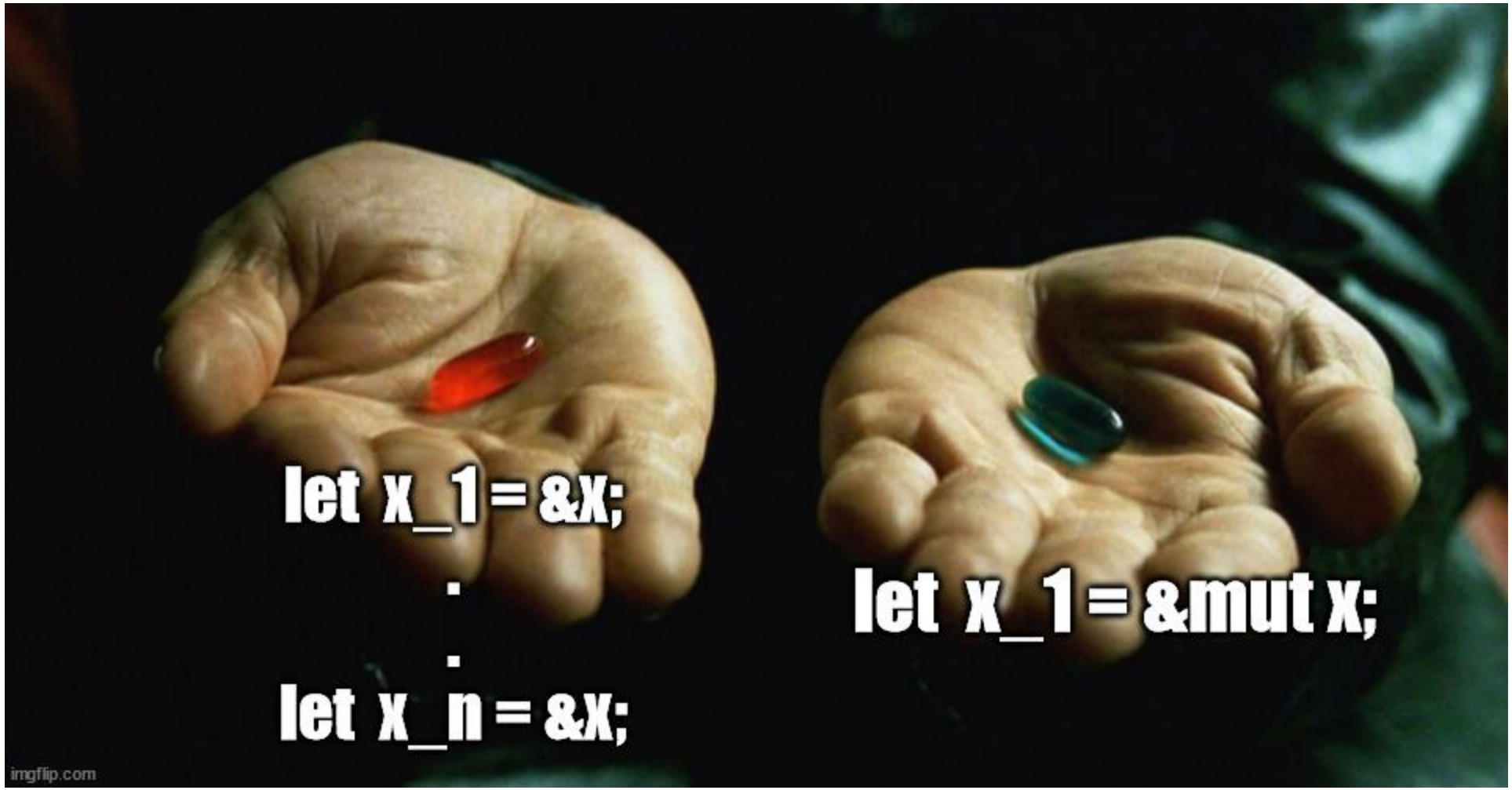
- You can have mutable references

```
let s1: &mut String = &mut s;
```

- However, at any point in time, you can either have:

Many immutable references **OR** 1 mutable (exclusive) reference

Borrowing



Borrowing

Example 4

```
let s: String = String::from("Hello world");

let s1: &String = &s;
let s2: &String = &s;    // OK, multiple immutable borrows

println!("{}, {}", s1, s2); // Hello world, Hello world
```

Example 5

```
let mut s: String = String::from("Hello world");

let s1: &mut String = &mut s;  // OK, only one mut
*s1 = String::from("Bye bye");  // Can mutate string

println!("{}", s);  // Bye bye
```

Borrowing

Example 6

```
let mut s: String = String::from("Hello world");

let s1: &String = &s;
let s2: &mut String = &mut s; // ERROR!!
*s2 = String::from("Bye world");

println!("{}", s1);
```

```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
  --> src\main.rs:30:14
   |
29 |     let s1 = &s;
   |              -- immutable borrow occurs here
30 |     let s2 = &mut s;  // ERROR!!
   |              ^^^^^^ mutable borrow occurs here
...
33 |     println!("{}", s1);
   |                    -- immutable borrow later used here
```

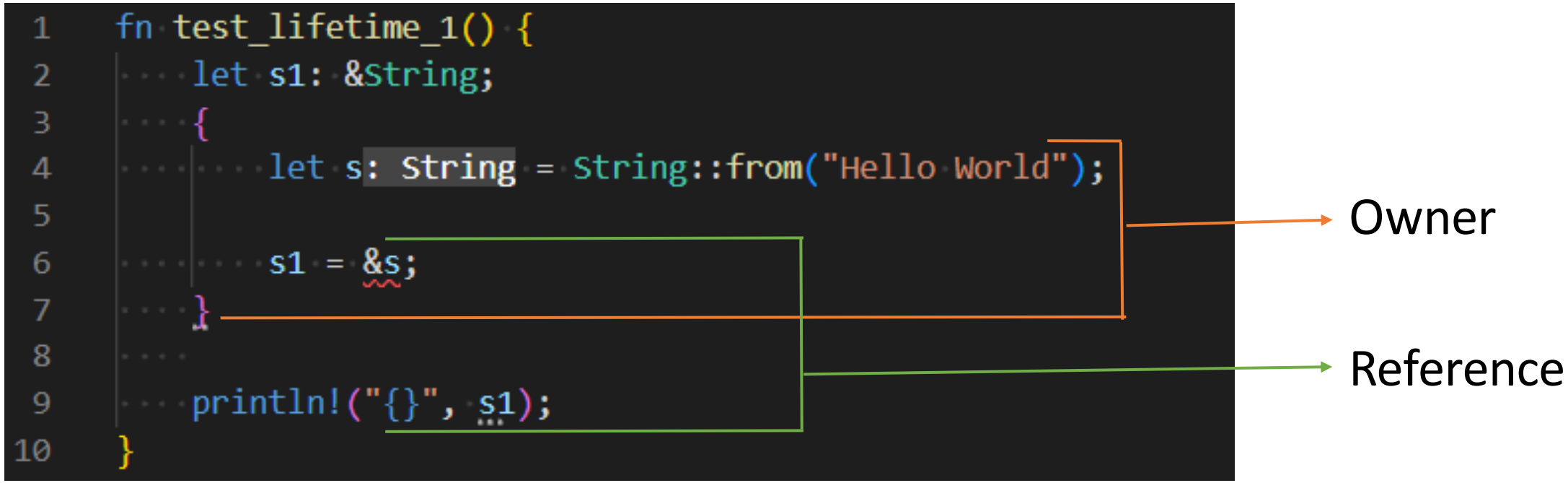
Lifetimes

```rust
let s1: &String;
{
    let s: String = String::from("Hello World");
    s1 = &s;
}
println!("{}", s1);
```

```
error[E0597]: `s` does not live long enough
 --> src\lifetime_examples.rs:5:14
  |
5 |         s1 = &s;
  |              ^^ borrowed value does not live long enough
6 |     }
  |     - `s` dropped here while still borrowed
7 |     println!("{}", s1);
  |                    -- borrow later used here

For more information about this error, try `rustc --explain E0597`.
error: could not compile `ownership` due to previous error
```

Lifetimes

- Lifetime = how long a reference lives (its scope)
- The lifetime of the borrower cannot outlive the borrowed value's owner



Going back to safety

```cpp
int main() {
    char *buffer = new char[42];
    delete [] buffer;

    // ...

    buffer[0] = 42;  // Boom!
    return 0;
}
```

```rust
let mut v: Vec<char> = Vec::with_capacity(42);
drop(v);

// ...
v[0] = 'c';
println!("{}", v[0]);
```

```
error[E0382]: borrow of moved value: `v`
 --> src\cpp_examples.rs:7:5
  |
3 |     let mut v: Vec<char> = Vec::with_capacity(42);
  |         ----- move occurs because `v` has type `Vec<char>`, which does not implement the `Copy` trait
4 |     drop(v);
  |          - value moved here
...
7 |     v[0] = 'c';
  |     ^ value borrowed here after move
```

Going back to safety

```
1  int* g(int* a, int* b) { return b; }
2
3  int main()
4  {
5      int x;
6      int* p = g(&x, nullptr);
7      *p = 42; // boom
8  }
```

- No "nullptr" in Rust
- Would fail because of lifetime check
- "p does not live long enough"

Going back to safety

```cpp
auto v = std::vector<int>();
v.push_back(0);
auto first_elem = &v[0];
v.push_back(1);
std::cout << *first_elem << std::endl;
```

```rust
let mut v: Vec<i32> = vec![];
v.push(0);
let first_elem: &i32 = &v[0];
v.push(1);
println!("{}", *first_elem);
```

```
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable
  --> src\cpp_examples.rs:43:5
   |
42 |     let first_elem = &v[0];
   |                       - immutable borrow occurs here
43 |     v.push(1);
   |     ^^^^^^^^^ mutable borrow occurs here
44 |     println!("{}", *first_elem);
   |                     ---------- immutable borrow later used here
```

Smart pointers

	C++	Rust
Unique reference	unique_ptr<T>	Box<T>
Shared reference	shared_ptr<T>	Rc<T>

```
let s: MyStruct = MyStruct {x : 10};
let a: Box<MyStruct> = Box::new(s);    // Now a points to somewhere in the heap
println!("{:?}", a);
```

```
let s: MyStruct = MyStruct {x : 10 };
let rc: Rc<MyStruct> = Rc::new(s);

let s1: Rc<MyStruct> = rc.clone();  // Increments reference count
let s2: Rc<MyStruct> = rc.clone();  // Increments reference count

println!("{:?}, {:?}", s1, s2);
```

Runtime borrow checking

- RefCell<T> is a smart pointer that checks borrows at runtime
- It can give mutable reference from immutable object (!!)
- This is known as "Internal Mutability Pattern"

```
let x: RefCell<MyStruct> = RefCell::new(MyStruct {x : 10});

// Returns &MyStruct. Panics if value is currently mutably borrowed
let x_ref: Ref<MyStruct> = x.borrow();

// Return &mut MyStruct. Panics if value is currently immutably borrowed
let x_ref_mut: RefMut<MyStruct> = x.borrow_mut();
```

Going multithreading

- A type is **Send** if it is safe to send it to another thread
- A type is **Sync** if it is safe to share between threads (T is Sync if and only if &T is Send)

	Single thread	Multiple threads
Sharing data	Rc<T>	Arc<T>
Allow mutability	RefCell<T>	Mutex<T>